Monads by Example

Izaak Weiss

Contents

1 What are Monads?

It is entirely reasonable that this is the first question that anyone learning Monads asks, and it is also entirely reasonable that anyone who is teaching Monads tries to answer it. However, Monads are a complex concept that cannot be explained in a single sentence or even a single paragraph; to understand

Monads you must simultaneously understand the problem they are trying to solve, their implementation, the interface for working with them, and the theoretical computational background. Therefore, I will not try and answer this question in a single phrase; my explanation of what Monads are is the entirety of this paper.

I would like to take a few moments and clear up one possible misconception. Monads are not special. They are a data structure, just like a Linked List or a Dictionary. They have methods that you can call, and they store data in the same way. They don't have a common sounding name, so they seem scary, and people have a tendency to define them using complex math or weird analogies, but I firmly believe that Monads aren't actually any more complicated than the run of the mill data structures that programmers use every day.

2 Our First Monads

In this section, we'll explore a common problem surrounding how to report errors to the caller of a function.

2.1 Error Handling in Plain Python

Python usually uses Exception raising and catching to report errors that happen within code. It's a desirable feature to have in a scripting language, but it is less useful in systems languages, like C, Go, or Rust, because exceptions are expensive in terms of memory and CPU time. It's also less useful in functional languages like Haskell or Scala, which use types and abstract data structures to make code more predictable and safer, a goal which is undermined when code can throw exceptions that crash the whole program. In the following section, I'll be exploring ways to handle errors in Python without throwing exceptions and without using Monads.

2.1.1 Division

```
def division(x, y):
    return x / y
```

Consider the above code fragment. This is a very simple function; one that is so simple it hardly deserves to exist. However, if `y` is zero, this function can throw a `ZeroDevisionError`. It's possible that we want to recover from this error gracefully; check the inputs and see if an error will occur, and return some error code instead of raising an exception.

However, we must decide what the error code should be. We can't choose `0.0`, because that is correctly returned by `division(0.0,1.0)`. We can't choose

`-1.0`, because that is correctly returned by `division(-1.0,1.0)`. In fact, this function can return any possible floating point number, so we can't choose a floating point number as our error code. One solution is to return a float on success, but `None` on failure.

```
def division(x, y):
    if y == 0:
        return None
    return x / y
```

Now we've written a function that checks whether or not division is possible, and performs division if it is, but returns an error code if it is not.

2.1.2 Indexing

```
def index(ls, i):
    return ls[i]
```

The above code is similar to the last example; it will perform an index lookup into a list, and return the item from the list if it can. If it can't, we're still left with the problem that it throws an exception if the index is out of bounds. Let's try and do the same thing as above; rewrite this function so that it doesn't throw an error, but instead uses an error code to signal something has gone wrong.

Our first guess might be to have our error code be the same as above, and just return `None`. However, this isn't possible, because the code `index([None], 0)` would also return `None`. In fact, python lets any value be inside of a list; there is no possible error code we can return that can't also be in the list. Luckily, python lets us use multiple return values.

```
def index(ls, i):
    if i < 0 or i >= len(ls):
        return True, None
    return False, ls[i]
```

This allows us to actually check whether or not this function has failed, without worrying about receiving an exception. This allows us to handle errors from outside of the function in a logical way:

```
failed, value = index([1,2,3],0)
if failed:
    print("Oh no, we failed")
else:
    print(value)
```

2.1.3 Combining The Above

```
def divide_elements(ls, i1, i2):
    return ls[i1]/ls[i2]
```

Now, we've combined the two operations in python which might lead to an exception, and we've done it in a way that allows for three different operations to result in an exception. We can rewrite this function so that it won't throw any errors, by checking after each step for an error code.

```
def index(ls, i):
    if i < 0 or i >= len(ls):
        return True, None
    return False, ls[i]

def division(x, y):
    if y == 0:
        return None
    return x / y

def divide_elements(ls, i1, i2):
    failure, v1 = index(ls, i1)
    if failure:
        return None

    failure, v2 = index(ls, i2)
    if failure:
        return None

    return division(v1, v2)
```

This code works nicely; you can throw two types of errors at it, and it returns `None` when either error would occur.

2.1.4 Other Programming Languages

Python has been very nice to us so far; in Python, it is easy to write a function that returns two values, or returns different types in different scenarios. Python also has other features, such as Exceptions, which make this rewriting we've been dong sort of useless. The most Pythonic way of writing the above would probably be to catch the exceptions, writing:

```
def divide_elements(ls, i1, i2):
    try:
        return ls[i1]/ls[i2]
```

```
    except (IndexError, ZeroDivisionError):
        return None
```

Other languages have their own ways of dealing with the errors we discussed above, and they all have their own benefits. In C, the common pattern is to have the actual return value of the function be a number that indicates whether an error occurred, and if one did, what the error was. To get the meaningful result from the function, you pass a pointer to a block of memory into the function, and that function writes the answer you want into that block of memory. This is much faster and simpler than having to write all of the infrastructure required to deal with throwing exceptions and allowing someone above you to catch that exception.

In many modern languages, including Rust, Scala, and Haskell, the solution of choice is to use Monads.

2.2 The Option Monad

The Option Monad, also called the Maybe Monad in many programming languages, is a way of representing the result of a function or computation that might result in an error and produce no meaningful output. We call an Option Monad that has a value and represents a successful computation Some, and we call one that doesn't have a value and represents a failed computation Nothing.

These are really easy to write in programming languages like Haskell, Scala, or Rust, but I'm going to write it in Python to help avoid any confusion about the inner workings of the Monad. In the end, all Monads are just objects, like trees, lists, or dictionaries.

```
class Option:
    def __init__(self, failed, value):
        self._failed = failed
        self._value = value

    def __repr__(self):
        if self._failed:
            return 'Nothing'
        else:
            return 'Some({})'.format(self._value)

    def is_some(self):
        if self._failed:
            return False
        return True

    def is_none(self):
```

```
        return not self.is_some()

    def unwrap(self):
        if self._failed:
            raise Exception('This Option has no value')
        else:
            return self._value

    @classmethod
    def some(cls, x):
        return cls(False, x)

    @classmethod
    def none(cls):
        return cls(True, None)
```

This is the longest piece of code we've had so far, so let me break it down bit by bit.

```
class Option:
    def __init__(self, failed, value):
        self._failed = failed
        self._value = value

    def __repr__(self):
        if self._failed:
            return 'Nothing'
        else:
            return 'Some({})'.format(self._value)
```

The `__init__` function is the constructor or initializer function for classes in python. Here all we do is create a boolean that indicates whether or not we have failed the computation or not, and if we haven't failed it, we store the result of the computation in `_value`. Note that if `_failed` is `True`, then we don't care what is in `_value`, because the computation has failed and that value has no meaning. I will note that the users of this class will probably never call `__init__` themselves, as we will later write alternate constructors that are easier for people to use.

The `__repr__` function simply tells python how this object should be printed.

```
    def is_some(self):
        if self._failed:
            return False
        return True

    def is_none(self):
```

```
        return not self.is_some()

    def unwrap(self):
        if self._failed:
            raise Exception('This Option has no value')
        else:
            return self._value
```

These three functions are the meat of the Option Monad; these are the ways we interact with it. The `is_some` function returns `True` when there is a meaningful return value, and `False` if the computation failed. `is_none` does the opposite. `unwrap` returns the value of the Option Monad *if there is a value to be returned*, otherwise it throws an error. In order to use the `unwrap` function without an error, you must first check and see whether the computation succeeded or failed.

```
    @classmethod
    def some(cls, x):
        return cls(False, x)

    @classmethod
    def none(cls):
        return cls(True, None)
```

These functions, decorated with `@classmethod`, aren't methods of the object. Instead, they're methods that exist as part of the class itself; here, we use them as alternate constructors.

At this point, let me rewrite our exception-free code from above using the Option Monad.

```
def division(x, y):
    if y == 0:
        return Option.none()
    return Option.some(x / y)

def index(ls, i):
    if i < 0 or i >= len(ls):
        return Option.none()
    return Option.some(ls[i])

def divide_elements(ls, i1, i2):
    res1 = index(ls, i1)
    if res1.is_none():
        return Option.none()

    res2 = index(ls, i2)
    if res2.is_none():
```

```
        return Option.none()

    return division(res1.unwrap(), res2.unwrap())
```

The above code is the exact same length in lines; and already has some benefits. First, these functions have a return type that can be determined just be looking at the code - not as useful in python as in other languages, but they are very useful in statically typed languages. Second, we don't have to remember the convention for every function. Before, we had to remember that `division` returned `None` for an error, but `index` returned `False, None` for an error. Despite these benefits, it's not that much of an improvement.

And that's because we haven't yet implemented the most important function for a Monad. This is the most important but also the most complicated part of the Option Monad, so I am going to insert a header for this.

2.3 The Bind and Fmap Functions

```
    def bind(self, function):
        if self.is_none():
            return self

        val = self.unwrap()
        return function(val)
```

The bind function is higher order function. That means it's a function that takes another function as an argument, and does something with that function. If you look in the end of the previous section, we repeated this piece of code two times:

```
    res = index(ls, i1)
    if res.is_none():
        return Option.none()
```

We were checking whether or not a function had successfully computed a value, or whether an error had occurred. If the value existed, we later passed that value into a function. If the value did not exist, then we simply returned the indication of failure, an `Option.none()` object.

Looking at bind, we can see it does exactly that. `res.bind(function)` checks whether or not `res` is a successfully computed value, in which case it passes that value into `function`, or if it is a failed computation, in which case it simply returns itself, passing the failed computation forward.

```
    def fmap(self, function):
        if self.is_none():
            return self
```

```
        val = self.unwrap()
        return Option.some(function(val))
```

`fmap` does something similar to `bind`; whereas the function passed to `bind` might fail, and therefore returns an Option Monad, a function passed to `fmap` will always succeed, and therefore the return value needs to be wrapped in an Option Monad again. `fmap` and `bind` serve very similar purposes, but they are useful in different contexts.

So, with `fmap` and `bind` in mind, let's consider the our code again.

```
def division(x, y):
    if y == 0:
        return Option.none()
    return Option.some(x / y)

def index(ls, i):
    if i < 0 or i >= len(ls):
        return Option.none()
    return Option.some(ls[i])

def divide_elements(ls, i1, i2):
    res1 = index(ls, i1)
    res2 = index(ls, i2)

    partial = lambda x: lambda y: division(x,y)

    return res2.bind(res1.bind(partial))
```

I've changed a few things, so go back and look at what I've done. There's this new line `partial = lambda x: lambda y: division(x,y)`. This is a way of defining an inline function in python; here, I have a function of x that returns a function of y, that itself returns `division(x,y)`. This makes it so that instead of calling this function with two arguments, I instead call it with one argument twice. Here's an example of using a similar lambda function:

```
partial = lambda x: lambda y: x/y

two_over = partial(2)
two_over(3) # same as 2 / 3
two_over(5) # same as 2 / 5

ten_over = partial(10)
ten_over(50) # same as 10 / 50

partial(3)(4) # same as 3 / 4
```

What I do on the return statement line is I use bind to apply the function to these arguments; the function `division` will fail drastically if it gets an Option Monad in as an argument; it would fail on the comparison to zero and it would also fail on trying to divide two Option Monads. By using `bind`, I am telling the Options to apply the function to themselves if they have a value, or returning Nothing if they don't have a value. This wraps the error handling into the data structure that represents errors without me having to ever write an explicit `if x.is_some():` check.

You may still think this sort of thing is useless; and in python, for such a simple example, it kinda is! But as we continue to explore Monads, we will encounter some examples of things that get more and more complex without Monads, but that Monads make simpler. Oh hey look, that's the next section.

2.4 A More Complex Example

In order to give a more illustrative example of where the Option Monad can be more useful, consider the following problem; open a file, and read the first whitespace separated word from the beginning of the file, and parse it into an integer if possible. This problem is fairly easy to do with built in Python functions, but the Option Monad can make error handling easier. However, none of Python's built in functions use the Option Monad, so we will have to rewrite them so that they do. In languages with the Option Monad as a star player, such as Rust, Haskell, or Scala, this isn't an issue.

```
def option_open(filename, mode='r'):
    try:
        fd = Option.some(open(filename, mode=mode))
    except Exception:
        fd = Option.none()
    return fd

def option_read(fd, size=-1):
    try:
        data = Option.some(fd.read(size))
    except Exception:
        data = Option.none()
    return data

import re

def option_match(pattern, string):
    match = re.match(pattern, string)
    if match:
        match = Option.some(match)
    else:
```

```
        match = Option.none()
    return match

def option_get_group(match, group):
    try:
        g = match.group(group)
    except Exception:
        g = None

    if g == None:
        g = Option.none()
    else:
        g = Option.some(g)

    return g

def option_int(s):
    try:
        i = Option.some(int(s))
    except Exception:
        i = Option.none()

    return i
```

These functions perform the exact same operations as their Python counterparts, but they return Some if the computation succeeds and Nothing if it fails, instead of throwing an error or using some other return code. This will allow us to use `bind` to chain these functions.

```
result = (Option.some('text.txt')
    .bind(option_open)
    .bind(option_read)
    .bind(lambda x: option_match(r'\s*(\S*)', x))
    .bind(lambda x: option_get_group(x, 1))
    .bind(option_int))
```

This code opens a text file, reads the entire file from it, looks at the first word in the file, and tries to read it in as an integer. Using the Option Monad is useful because if an error happens at any time during the computation, it just passes a Nothing Option through the rest of the bind functions.

We can make this look cooler by choosing an infix operator to overload. By convention, `>>=` is used, but that's hard to to do in python, so we are going to use `>>`. We can override that operator in python with the following code:

```
def __rshift__(self, function):
    return self.bind(function)
```

Now, let's rewrite the above option code as the following.

```
result = (
    Option.some('text.txt')
      >> option_open
      >> option_read
      >> (lambda x: option_match(r'\s*(\S*)', x))
      >> (lambda x: option_get_group(x, 1))
      >> option_int
    )
```

Consider the same operation in regular Python. We can write it in one expression, in which case this is impossible to read, or we can split it up, over many lines, creating a bunch of temporary variables that we use once and then never again.

```
# one expression
result = int(
        re.match(
            r'\s*(\S*)',
            open('text.txt').read()
        ).group(1)
    )

# with temporary variables
temp1 = open('text.txt').read()
temp2 = re.match(r'\s*(\S*)', temp1).group(1)
result = int(temp2)
```

The first example is unreadable. The functions used have no meaningful order, so it becomes an act of mental gymnastics to figure out what happens when. The functions appear in the order `int`, `match`, `open`, `read`, and `group`. `int` comes first, despite being called last, and `open`, the first function to be called, appears randomly in the middle.

The second example is the shortest version where all the functions appear in the source code in the order they are called, and so it is pretty readable, but once again we've got the problem of errors!

This code could blow up if the wrong information is passed into it, and it can blow up in approximately 5 places. Even worse, because `.group()` can return `None` if it fails, and `int(None) == 0`, you can get a wrong answer from this code without an error being thrown. In order to guarantee that this code doesn't fail, this is the sort of code you would need to write in Python.

```
try:
    temp1 = open('text.txt').read()
    temp2 = re.match(r'\s*(\S*)', temp1).group(1)
    if temp2 == None:
        result = None
```

```
    else:
        result = int(temp2)
except Exception:
    result = None
```

The above code now won't throw any errors or produce erroneous results, and will set `result` to `None` if the code fails. Now, compare that safe version without Monads to the version that is safe with Monads.

```
result = (
    Option.some('text.txt')
      >> option_open
      >> option_read
      >> (lambda x: option_match(r'\s*(\S*)', x))
      >> (lambda x: option_get_group(x, 1))
      >> option_int
    )
```

The Monad version is simpler, just as safe, and even a line shorter (two if you don't count the line with a single closing parenthesis). Not to mention cooler by far.

2.5 The Result Monad

There is one major problem with the Option monad above; if our code fails, we have no way to know how or when. With the standard python example, we could print an error message to the screen or to a file that would let us know what kind of error occurred. There is another Monad, called the Result Monad, that allows us to do that while still having the power of the Option Monad. I won't reproduce the entire code here (it is in the appendix) but I will go over a few of the changes, as we will use the Result Monad in the next section.

The Result Monad uses slightly different names; a value is Ok if it is a successful computation, and it is an Error if it has failed. The functions to check this status are `self.is_ok()` and `self.is_error()`.

```
    def __init__(self, failed, value, message):
        self._failed = failed
        self._message = message
        self._value = value
```

Our `__init__` function now takes an additional argument; an error message. Now, we have a value that indicates whether or not our computation has failed, a value that stores the result of the computation (if the computation succeeded), and a value that stores the error message (if the computation failed).

In order to access the error message, we add a new function like `unwrap`.

```
    def error_msg(self):
        if self.is_error():
            return self._message
        else:
            raise Exception('This Result is Ok')
```

This function checks whether or not we have failed, and returns the error message if it is an Error. Just like `unwrap`, it throws an Exception if there is no error message.

`bind` has not changed at all, but it is nice to note that when you return `self` in the case of the error, the error message stays the same.

```
    def bind(self, function):
        if self.is_error():
            return self

        val = self.unwrap()
        return function(val)
```

I'm also adding another function that is sort of like `bind`, but instead, provides a simple way for Monadic computations to check for errors, and if they've occurred, to replace the computed value with a default value.

```
    def recover(self, function):
        if self.is_error():
            return function()

        return self
```

This function allows you to perform an operation like getting a value from a configuration file, but using a default value if it fails, with Monads. The following snippet, for example, uses results to open, read, and parse some number from a file, and then it checks whether the operation failed at any level, and uses the value of `10` if it failed.

```
config = (
    result_open('config.txt')
      >> result_read
      >> result_parse
    ).recover(lambda: Result.ok(10))
```

The equivalent in plain Python would be the following.

```
try:
    temp1 = open('config.txt').read()
    config = parse(temp1)
except Exception:
    config = 10
```

We've already seen these used above, but for completeness here are the two constructors for the Result Monad.

```
    @classmethod
    def ok(cls, val):
        return cls(False, val, None)

    @classmethod
    def error(cls, msg):
        return cls(True, None, msg)
```

3 A Parsing Monad

We're now going to talk about a Monad called the Parsing Combinator, which basically recreates and improves upon regular expressions using Monads. Here we see the truth of the oft said but little understood aphorism that Monads represent computation; here, they represent the computation of regular expressions.

It's also worth noting that there are many python Parsing Combinator libraries on the python package index, and those will have faster, more powerful implementations than the one below.

3.1 The Code

```
class Parser:
    def __init__(self, function):
        self._function = function

    def __call__(self, text, index):
        return self._function(text, index)

    def __repr__(self):
        return '<Parsing Combinator>'
```

The basic idea behind our Parsing Combinator is that the Monad holds a function that takes a string, which we're going to call `text`, and it returns a Result Monad. If the result is a success, the Result Monad holds a tuple of the text that has matched the current parser, and the remainder of `text`. In order to make this a bit easier for us, we will implement the special method `__call__`, so that we can write stuff like `self(text)` instead of `self._function(text)` in the future. It's really hard to print functions meaningfully, so we're just going to define our representation as a constant string.

I'm going to slightly deviate from our already established order to introduce the

constructors first; then we can deal with the ways we can bind to and combine Parsing Combinators. These constructors will do two things; define a function, and then create a Parser using that function.

```
    @classmethod
    def char(cls, val):

        def match_char(text):
            try:
                current = text[0]
            except IndexError:
                return Result.error('End of String encountered'
                    ', but {} is still expected'.format(repr(val)))

            if current == val:
                return Result.ok((text[0], text[1:]))
            else:
                return Result.error('Failed to match character' +
                    ' {} at {}'.format(repr(val), repr(text)))

        return Parser(match_char)

    @classmethod
    def empty(cls):

        def match_empty(text):
            return Result.ok(('', text))

        return Parser(match_empty)

    @classmethod
    def oneof(cls, charls):

        def match_charls(text):
            try:
                current = text[0]
            except IndexError:
                return Result.error('End of String encountered, but '
                    'one of {} is still expected'.format(list(charls)))

            if current in charls:
                return Result.ok((text[0], text[1:]))
            else:
                return Result.error('Failed to match one of {} at {}'
                    .format(list(charls), repr(text)))
```

```
        return Parser(match_charls)

    @classmethod
    def noneof(cls, charls):

        def none_charls(text):
            try:
                current = text[0]
            except IndexError:
                return Result.error('End of String encountered,'
                    ' but none of {} is still expected'.format(repr(text)))

            if current not in charls:
                return Result.ok((text[0], text[1:]))
            else:
                return Result.error('Found one of'
                    ' {} at {} when there should be none of'.format(
                    list(charls), repr(text)))

        return Parser(none_charls)
```

These all follow fairly simple rules; they are the equivalent of some fairly basic regular expressions. The first one tries to compare the first character of the text to a character, and if it can, it pulls that character off of the text, and returns the character and the remainder as the result. The second one does absolutely nothing to the input text, simply returning the empty string and the input unchanged. The third and fourth represent character classes, but still match exactly one character from the beginning of the text. To do advanced parsing, we're going to need some more advanced constructors; ones that combine other parsers into new parsers.

```
    def combine(self, other, function):

        def combine_func(match, rest):
            res = other(rest)
            if res.is_ok():
                other_match, rest = res.unwrap()
                new_match = function(match, other_match)
                return Result.ok((new_match, rest))
            else:
                return res

        return Parser(lambda text: self(text)
            .bind(lambda res: combine_func(*res)))
```

`combine` is the most useful function for us; it takes two parsers (`self` and `other`)

and a function. It creates a new parser that first executes `self` on the input text, and then it executes `other` on the remaining text after `self`'s match. Then it uses `function` to combine the two matches, and returns that combination along with the remaining text. We're also going to define a few functions that call combine with a particular function

```
    def concat(self, other):
        return self.combine(other, lambda x,y: x + y)

    def first(self, other):
        return self.combine(other, lambda x,y: x)

    def last(self, other):
        return self.combine(other, lambda x,y: y)

    def tuple(self, other):
        return self.combine(other, lambda x,y: (x,y))
```

These functions are the most common choices for what you might want to use to combine the two matches. Most common is `concat`, which simply concatenates the two strings; this is the default regex behavior. Then there is `first` and `last`, which instead of combining the two values, instead discard one of the values. We will see later why this is useful. Finally, we can put the two values into a tuple instead of concatenating them. This is mostly useful when used in conjunction with `bind`, which we will discuss later. However, I've provided a few examples of working with these functions.

```
match_a = Parser.char('a')
match_b = Parser.char('b')

# This will match the string 'ab' and return the string 'ab'
match_a.concat(match_b)

# This will match the string 'ab' and return the string 'a'
match_a.first(match_b)

# This will match the string 'ab' and return the string 'b'
match_a.last(match_b)

# This will match the string 'ab' and return the tuple ('a','b')
match_a.tuple(match_b)
```

We also have a few other important functions that we use to build more complex parsers.

```
    def choice(self,other):
```

```
        def choice_func(text):
            return self(text).recover(lambda: other(text))

        return Parser(choice_func)
```

`choice` implements the ability to try one parser, and if it fails, recover by trying another. Be careful with this one, because it only operates locally. If you try and match **x** or **y**, and **x** succeeds in matching the text but puts you in a corner that causes failure later on, it won't backtrack and try **y**. It will only backtrack and try **y** if matching **x** fails.

```
    def many(self, function=lambda x,y: x + y):

        def repeat_func(text):
            res = self(text)

            if res.is_error():
                return Result.ok(('',text))

            match = res.unwrap()[0]
            rest = res.unwrap()[1]

            res = self(rest)

            while res.is_ok():
                match = function(match, res.unwrap()[0])
                rest = res.unwrap()[1]
                res = self(rest)

            return Result.ok((match, rest))

        return Parser(repeat_func)

    def many1(self, function=lambda x,y: x + y):
        return self.combine(self.many(function), function)
```

`many` is perhaps the most complicated constructor. What we want to do is continually match one parser, using a function to combine the results, until a failure occurs; but when a failure occurs, we want to ignore the failure and return that previous match. We also have a useful `many1` function, which is the same as `many` but demands that at least one match succeeds.

```
    def optional(self):
        return self.choice(Parser.empty())
```

Finally, `optional` tries to match the input text with `self`, but if it fails, it matches nothing and finishes.

Finally, we add symbolic versions of many of the above functions. This is purely for ease of reading the expressions we will write; you will see that they can get pretty complex, and `this.concat(that)` is harder to understand at a glance than `this + that`.

```
    # |
    def __or__(self, other):
        return self.choice(other)

    # +
    def __add__(self, other):
        return self.concat(other)

    # =>
    def __ge__(self, other):
        return self.last(other)

    # <=
    def __le__(self, other):
        return self.first(other)

    # &
    def __and__(self, other):
        return self.tuple(other)
```

Now we can move on to the Monadic functions.

```
    def bind(self, function):

        def bind_func(result):
            function(result[0]).bind(lambda x:
                Result.ok(x, result[1]))

        return Parser(lambda text: self(text).bind(bind_func))
```

`bind` is, of course, the star of the show. Lets break this down into steps, starting from the the return statement and moving back.

- We're creating a new Parser, so we're passing in a function that takes text, and should return a Result Monad holding `(matched value, remainder of text)`.
- The first thing this function does is pass `text` through the current parser's function. This should result in a Result Monad holding `(matched value, remainder of text)`.
- Then, we call bind on that result, passing in `bind_func`. `bind_func` is going to get a tuple as it's only argument, and should return a Result Monad holding `(matched value, remainder of text)`.
- `bind_func` takes the result, and passes the matched value through the

function passed in to the Parser's bind function. This gives us a Result Monad, and we pull out it's value (if it exists).
- Then we build the new Result Monad holding `(matched value, remainder of text)`, except now the matched value is the result of `function`.

EXPLAIN WHAT BIND DOES AT A HIGH LEVEL

```
    def fmap(self, function):
        return self.bind(lambda x: Result.ok(function(x)))
```

We're also going to introduce a new function very similar to `bind` called `fmap`. This function is the same as `bind`, but it takes a function that can't fail and uses that to modify the matched value. We could always write out `bind(lambda x: Result.ok(function(x)))` for this, but it's nice to have it packaged up in a method, because then we can bind it to a symbol.

```
    # >>
    def __rshift__(self, function):
        return self.bind(function)

    # >
    def __gt__(self, function):
        return self.fmap(function)
```

Finally, let's write some functions that actually call the parser on some strings.

```
    def parse_prefix(self, string):
        return self(string)

    def parse_total(self, string):

        def check_full(tup):
            if tup[1] == '':
                return Result.ok(tup[0])
            else:
                return Result.error('The match did not consist '
                    'of the entire string: {} was left over'.format(
                    repr(tup[1])))

        return self(string) >> check_full
```

These two functions help make it easier to use the parser; instead of calling the parser ourself, we use these functions to parse either a prefix of the string using our parser, or the entire string.

3.2 Using the Parser Combinator

Now we have a powerful enough Parser Monad to recreate all of the flexibility and power of regular expressions. We can combine parsers to make more complex ones, we can define functions that return parsers that recreate common regular expressions syntax, so let's try parsing something simple; let's try and write a Parser Combinator that parses numbers.

A number can be as simple as `12` or as complex as `12.00123e45`, so we're going to need to build up a complex parser. Let's start with creating a parser that parses 1 or more of consecutive digits.

```
digits = Parser.oneof('0123456789').many1()
```

Now, we need to express an optional decimal place, followed by one of more digits.

```
decimal = (Parser.char('.') + digits).optional()
```

Now, we need an exponent part, which is pretty simple given the above. However, we do need one additional component, a sign.

```
sign = (Parser.char('+') | Parser.char('-')).optional()
exponent = ((Parser.char('e') | Parser.char('E'))
             + sign + digits).optional()
```

Finally, we can put these three together to get:

```
number = sign + digits + decimal + exponent
```

Here's our finished code, and it's results on some possible inputs:

```
digits = Parser.oneof('0123456789').many1()
decimal = (Parser.char('.') + digits).optional()
sign = (Parser.char('+') | Parser.char('-')).optional()
exponent = ((Parser.char('e') | Parser.char('E'))
            + sign + digits).optional()
number = sign + digits + decimal + exponent

Parser.parse_total(number, '12')
    # Ok('12')
Parser.parse_total(number, '12e10')
    # Ok('12e10')
Parser.parse_total(number, '2.12345e100')
    # Ok('2.12345e100')
Parser.parse_total(number, 'hello world')
    # Error(Failed to match one of
    # ['0', '1', '2', '3', '4', '5', '6', '7', '8', '9'])
Parser.parse_total(number, '99 bottles of beer on the wall')
```

```
    # Error(The match did not consist of the entire string:
    # ' bottles of beer on the wall' was left over)
Parser.parse_total(number, '')
    # Error(End of String encountered)
```

Furthermore, we can use a result-oriented version of python's `float` function, along with our `bind` operator, to cast these results to be floats instead of strings.

```
def result_float(x):
    try:
        return Result.ok(float(x))
    except Exception:
        return Result.error('Failed to cast to a float')

Parser.parse_total(number, '7.22345e10') >> result_float
    # Ok(72234500000.0)
```

3.3 Going Further

Everything we've done so far can basically be done in the exact same way by regular expressions. However, because of the way we've defined the Parsing Combinator, we can use `bind` and `fmap` inside of Parsing Combinator expressions. This allows us to parse complex expressions into fully formed and entirely arbitrary python objects.

```
number = (sign + digits + decimal + exponent) >> result_float
```

As above, we can put the cast to a float inside of the number parser, and it will automatically do the conversion whenever something is parsed. We can also take this further:

```
whitespace = Parser.oneof(' \n').many1()

# without infix operators
many_numbers = (
    (whitespace.optional().last(number)).fmap(lambda x: [x])
    ).many()

# with infix operators
many_numbers = (
        (whitespace.optional() >= number) > (lambda x: [x])
    ).many()
```

Let me take this text and expand it into English. This reads 'If there's any whitespace, match it and discard it, and parse a number following it. Then, put that number in a list, and repeat until parsing fails. Then combine all of the

lists (via concatenation), and return that.' And this works perfectly, 100% in pure standard python, with no `eval` or `exec` statements. And you absolutely cannot do this with regular expressions.

```
text = '2.12345e+100 2.1e10 1223 13.5 100e100'
print(Parser.parse_total(many_numbers, text))
# Ok([2.12345e+100, 21000000000.0, 1223.0, 13.5, 1e+102])
```

Parsing Combinators can be used to write powerful, modular, readable, and concise parsers for any format of text. For example, here is a CSV parser in 5 lines of code.

```
expression = Parser.noneof(',\n').many1()
comma = Parser.char(',')
newline = Parser.char('\n')
line = ((expression <= comma.optional()) > lambda x: [x]).many()
csv = ((line <= newline.optional()) > (lambda x: [x])).many()

text = '''1,2,3,4,5
hello world, my, good, friends, 5
0,1,2,3,4'''

print(csv.parse_total(text))
# Ok(
#     [
#         ['1', '2', '3', '4', '5'],
#         ['hello world', ' my', ' good', ' friends', ' 5'],
#         ['0', '1', '2', '3', '4']
#     ]
# )
```

And finally, here is a full arithmetic expression parser. This is more powerful than the power of regular expressions to break a string into tokens. It constructs the full AST from a purely textual input, just using a Parser Combinator. You'll notice that at this point, I need to define a function and pass that in to a Parser; we need to do this because this is a recursive expression we are parsing. However, it doesn't actually add to the complexity too much.

```
from collections import namedtuple
from enum import Enum, auto

class Op(Enum):
    PLUS = auto()
    MINUS = auto()
    TIMES = auto()
    DIV = auto()

Expr = namedtuple('Expr', ['Op','e1','e2'])
```

```
openp = Parser.char('(') + whitespace.optional()
closep = whitespace.optional() + Parser.char(')')
plus = whitespace.optional() + Parser.char('+')
        + whitespace.optional()
minus = whitespace.optional() + Parser.char('-')
        + whitespace.optional()
times = whitespace.optional() + Parser.char('*')
        + whitespace.optional()
div = whitespace.optional() + Parser.char('/')
        + whitespace.optional()

Plus = lambda x: Expr(Op.PLUS, x[0], x[1])
Minus = lambda x: Expr(Op.MINUS, x[0], x[1])
Times = lambda x: Expr(Op.TIMES, x[0], x[1])
Div = lambda x: Expr(Op.DIV, x[0], x[1])

def expr(text):
    # rec stands for recursive
    rec_plus = ((openp >= Parser(expr))
                & (plus >= Parser(expr))) <= closep
    rec_minus = ((openp >= Parser(expr))
                & (minus >= Parser(expr))) <= closep
    rec_times = ((openp >= Parser(expr))
                & (times >= Parser(expr))) <= closep
    rec_div = ((openp >= Parser(expr))
                & (div >= Parser(expr))) <= closep

    full = (
                (rec_plus > Plus)
                | (rec_minus > Minus)
                | (rec_times > Times)
                | (rec_div > Div)
                | number
            )

    return full(text)

text = '((1+2) * (9 - 11))'

print(Parser(expr).parse_total(text))
# Ok(
#     Expr(
#         Op=<Op.TIMES: 3>,
#         e1=Expr(
#             Op=<Op.PLUS: 1>,
```

```
#             e1=1.0,
#             e2=2.0
#         ),
#         e2=Expr(
#             Op=<Op.MINUS: 2>,
#             e1=9.0,
#             e2=11.0
#         )
#     )
# )
```

4 Theory of Monads

Now that we've seen a few examples of what a Monad is, we can talk about the formal definition. This is going to be the most abstract section of the text, but I'll try and keep any statements from category theory or abstract algebra from appearing here.

4.1 Defining Monads

Monads are a special type of object that contains with in it a value and a context. Our Option and Result Monads contained the result of a computation, along with contextual information regarding whether the operation had succeeded or failed. This allowed us to write programs that could detect failure elegantly. Our Parsing Monad contained the result of the parsing so far, as well as the rest of the text remaining to be parsed. Whenever you see a Monad, you can sum up its operation by asking "What is the value in this Monad, and what is the context?". Many people, when confronted with Monads, want a way to get the value out of the Monad. But this causes problems, because you've taken the value out of context, and it becomes significantly more useless.

In order to interact with the values without taking them out of their context, we have a function called `fmap`. `fmap` takes the value, and applies the function to that value, and puts the result of he function back into context. In our Option and Result Monads, it applied the function to the value if our computation was successful, or it simply bypassed the function if our computation had failed. In our parsing combinator, it applied the function to the result of our computation, while leaving the remainder of the text to be parsed alone.

However, this meant that we couldn't use functions on the values in our Monad if the functions themselves returned Monads. For our Option and Result Monads, that means that we couldn't use a function that could fail (a function that returned an Option or Result Monad) on our Monad with `fmap`. If we did that,

we could end up with a recursive Monad, like `Some(None)` or `Some(Some(3))`. This is annoying, and there's two ways to fix this weirdness.

The first way, which we used in the previous section, is to use a function called `bind`. `bind` is the same as `fmap`, but instead of putting the return value of the function passed to `bind` back into the same context, it expects that `bind` will return a new value and context (a Monad of some kind).

The second way is an alternative to `bind`; you don't need both, and it's more common to have `bind` as the one to use, so I haven't bothered talking about it yet. This function is called `join`, and it takes a recursive Monad and flattens it from two layers to one layer. For example:

```
Option.some(Option.some(x)).join() == Option.some(x)
Option.some(Option.none()).join() == Option.none()
Option.none().join() == Option.none()
```

We can show that `join` isn't any less useful than `bind` by actually writing `bind` using only `join` and `fmap`.

```
def bind(monad, function):
    return monad.fmap(function).join()
```

This does the same thing as `bind` usually does; it applies the function to the inside value if the Monad isn't Nothing, and then it returns Nothing if either the function returns or the Monad is Nothing, or it returns Some(value) if the function succeeds and the Monad had a value to pass into the function.

We can also show a way to write `fmap` and `join` solely using `bind`:

```
def fmap(monad, function):
    monad.bind(lambda x: Option.some(function(x)))

def join(monad):
    monad.bind(lambda x: x.unwrap() if x.is_some() else x)
```

This means, for our purposes, for something to be a Monad, we require it to either have both `fmap` and `join` *or* `bind`.

4.2 Monad Laws

Now, Monads have three laws, or rules, they have to follow; this is just to make sure Monads don't have any unexpected behavior, but we should go over those rules anyway.

First of all, if you `fmap` over a Monad with the identity function (a function that returns its inputs unchanged), the value in the Monad doesn't change.

```
Monad(x).fmap(lambda x: x) == Monad(x)
```

Secondly, if you `fmap` two functions in a row, it should be the same as simply `fmap`ing the function which does the equivalent of those two functions in order.

```
m.fmap(lambda x: x+1).fmap(lambda x: x+2) == m.fmap(lambda x: x+3)
```

Finally, if you apply a function to a value and then stick it in a Monad, it is the same as putting that value in a Monad and `fmap`ing that function.

```
Monad(x).fmap(f) == Monad(f(x))
```

If you want, you can put these rules into equivalent forms using `bind` instead of `fmap`.

These might seem common sense, and if they are, that's good! The only reason that we require that these rules are followed is so that somebody doesn't create a Monad that behaves weirdly and it screws up our program. If they don't seem common sense, that's okay; you don't really have to understand them.They basically boil down to "Monads should behave sensibly when you fmap functions over them".

5 The Zeroth Monad

Our first section was titled 'Our First Monad'. However, we are computer scientists, and therefore we start counting at zero, not at one. So let's talk about another Monad that everyone reading this document has probably used, but never noticed that it was a Monad. It turns out that a list is a Monad.

How is a list a Monad? Well, from the previous section, a Monad is really just anything with a `bind` function, or with a `fmap` and a `join` function. And while not every programming language has these functions built in, we can easily write these functions for a list.

```
# python has a built in function, 'map' that does this.
def fmap(ls, function):
    new = []
    for item in ls:
        new.append(function(item))
    return new

# this is sometimes called 'flatten'
def join(ls):
    new = []
    for sublist in ls:
        for item in sublist:
            new.append(item)
```

```
    return new

# bind can be defined entirely with the other two!
def bind(ls, function):
    return join(fmap(ls, function))
```

This is all nice and well that we now have these functions, but it explains little conceptually. So let's try and describe Monads conceptually, and see how that can be applied to lists.

Our first Monads, the Option Monad and the Result Monad, both represented some sort of computation result that required more context than a simple value; in particular, they represented a computation result that could either succeed, producing a value, or fail, producing no meaningful value.

Lists can be thought of in a similar way; instead of representing either zero or one meaningful return value, lists can represent computations that can return zero, one, or any possible number of return values. For example, consider the following contrived example.

```
def less_than_abs(x):
    if x == 0:
        return []
    ls = [0]
    for i in range(1,x):
        ls.append(i)
        ls.append(-i)
    return ls

from math import sqrt

def sqrts(x):
    if x < 0:
        return []
    elif x == 0:
        return [0.0]
    else:
        return [sqrt(x), -sqrt(x)]

bind(
    bind(
        [3],
        less_than_abs
    ),
    sqrts
)
# [0.0, 1.0, -1.0, 1.4142135623730951, -1.4142135623730951]
```

In this case, we execute two functions in series, getting all of the valid results to our question in one list; but the number of results isn't the same for all inputs, so we need a Monad to represent this computational uncertainty.

A Code: The Result Monad

```
class Result:
    def __init__(self, failed, value, message):
        self._failed = failed
        self._message = message
        self._value = value

    def __repr__(self):
        if self._failed:
            return 'Option.error({})'.format(repr(self._message))
        else:
            return 'Option.ok({})'.format(repr(self._value))

    def __str__(self):
        if self._failed:
            return 'Error({})'.format(self._message)
        else:
            return 'Ok({})'.format(self._value)

    def is_ok(self):
        if self._failed:
            return False
        return True

    def is_error(self):
        return not self.is_ok()

    def unwrap(self):
        if self.is_ok():
            return self._value
        else:
            raise Exception('This Result is an Error')

    def error_msg(self):
        if self.is_error():
            return self._message
        else:
            raise Exception('This Result is Ok')

    def bind(self, function):
        if self.is_error():
            return self

        val = self.unwrap()
```

```
        return function(val)

    def fmap(self, function):
        if self.is_error():
            return self

        val = self.unwrap()
        return Result.ok(function(val))

    def recover(self, function):
        if self.is_error():
            return function()

        return self

    def __rshift__(self, function):
        return self.bind(function)

    @classmethod
    def ok(cls, val):
        return cls(False, val, None)

    @classmethod
    def error(cls, msg):
        return cls(True, None, msg)

# The following are built in functions
# rewritten to work with the Result Monad

def result_open(filename, mode='r'):
    try:
        fd = Result.ok(open(filename, mode=mode))
    except Exception:
        fd = Result.error("Failed to open the file")
    return fd

def result_read(fd, size=-1):
    try:
        data = Result.ok(fd.read(size))
    except Exception:
        data = Result.error("Failed to read from the file")
    return data

import re
```

```
def result_match(pattern, string):
    match = re.match(pattern, string)
    if match:
        match = Result.ok(match)
    else:
        match = Result.error("Failed to match the pattern")
    return match

def result_get_group(match, group):
    try:
        g = match.group(group)
    except Exception:
        g = None

    if g == None:
        g = Result.error("Failed to get the group from the match")
    else:
        g = Result.ok(g)

    return g

def result_int(s):
    try:
        i = Result.ok(int(s))
    except Exception:
        i = Result.error("Failed to parse into an integer")
    return i


result = (
    Result.ok('text.txt')
      >> result_open
      >> result_read
      >> (lambda x: result_match(r'\s*(\S*)', x))
      >> (lambda x: result_get_group(x, 1))
      >> result_int
    )

print(result)
```